DART/HYESS Users Guide

Jerome H. Friedman
Department of Statistics and
Stanford Linear Accelerator Center
Stanford University
jhf@stat.stanford.edu

December 20, 1996

Abstract

This note provides information for using the Fortran program [Friedman (1996b)] that implements the recursive covering approach to local learning described in Friedman (1996a).

1. Introduction

DART/HYESS is a collection of Fortran subroutines [Friedman (1996b)] that implement the recursive covering strategy described in Friedman (1996a). A familiarity with that paper is assumed. With this implementation the user writes a main program that reads in and stores the training data as well as declaring (additional) arrays that serve as output and scratch workspace for various subroutines that are called to perform training and prediction. The particular subroutines that are called from the main program depend upon the strategy used to accomplish these tasks. The principal subroutines are:

DART: purely eager training. Constructs and returns a complete cover tree to be used for prediction.

ANS & AVE: use the cover tree produced by DART to make individual output predictions.

START: hybrid (eager/lazy) training/prediction. Builds and returns a partial cover tree to be used in conduction with HYESS for making predictions.

HYESS: uses the cover tree produced by START to make individual output predictions.

XVALID: uses cross-validation to estimate the optimal number of observations K in each terminal region.

In addition to these routines that perform computation, there are a variety of others that can be called to change the values of various parameters that control the details of the learning strategy.

2. Eager training

With purely eager training all of the regions of the final cover are produced in advance of making any predictions. All information necessary for prediction is encoded into the representing data structure (cover tree) and the original training data are no longer needed. One advantage of this approach is very rapid prediction (ANS). Another is that information from more than one region can be combined to make individual predictions (AVE). The disadvantage is training time. Available computing generally limits the number of regions that can be produced to many fewer than can be accommodated with the lazy/hybrid strategy (see Section 3).

Eager training is accomplished by the subroutine call

$$call\ dart(no, ni, x, y, w, mxm, mxt, itr, rtr, trm, ms)$$

Input:

no = number of training observations.

ni = number of input variables.

$x(no, ni+3)$ = input data and workspace.

$x(no, 1 \rightarrow ni)$ = input variable data matrix.

$x(no,\ ni+1 \rightarrow ni+3)$ = scratch workspace.

$y(no)$ = corresponding training output variable values.

$w(no)$ = weight (mass) assigned to the respective training observations.

mxm = user set dimension of itr and rtr (see below).

mxt = user set dimension of trm (see below).

Output:

$itr(2, mxm)$, $rtr(3, mxm)$ = resulting cover tree.

$trm(mxt)$ = parameters of the local (terminal) models.

Workspace:

$ms(\,(no+1)\cdot(2\cdot ni+1)+1)$ = integer scratch array.

The first two quantities in the calling sequence (no, ni) define the size of the training data set. The next three x, y, w, are the training data which must be stored in the calling program before the call to DART. The next two itr, rtr store the cover tree upon return from DART. The array trm stores the parameters of the local approximators corresponding to each terminal node. The quantities mxm, mxt are the dimensions of these arrays as declared in the calling program; these values are used by DART to terminate execution if this storage is exceeded during the construction of the cover tree. If this happens a corresponding diagnostic message is printed, and the program must be rerun with increased declared storage. The minimum value of mxm depends on the number of terminal nodes mtr in the realized cover tree

$$mtr \simeq (no/kd)^{\log 2/\log(1-dtf)^{-1}}, \tag{2.1}$$

and on the value of lin (see below); if $lin = 0$ then $mxt > 2\cdot mtr$, otherwise $mxt > (ni+3)\cdot mtr$. The quantities kd and dtf are described below. A reasonable strategy is to make these dimensions as large as can be reasonably accommodated with the computer upon which the program is being executed. If this storage is exceeded during tree construction a diagnostic message will be issued and execution stops. If it is not possible to increase the dimension of the offending array, the size of the cover tree can be reduced by altering the value(s) of kd and/or dtf to produce a smaller tree (2.1). Alternatively, one can adopt the lazy/hybrid implementation described in Section 3.

After the call to DART (upon successful completion) the actual storage values needed by DART (for the particular problem) can be obtained by the subroutine call

$$call\ dtsize(mtr, mtm)$$

Output:

mtr = number of terminal nodes in the cover tree.

mtm = terminal node storage requirement. Minimum required dimension of trm array.

2.1. Parameters

There are a variety of parameters that control the strategy for building the cover tree [Friedman (1996a)]. They are set to default values in the program. These values can be changed by appropriate subroutine calls (with the new values as argument) before calling DART. These routines are:

call $setdtn(kd)$: kd = number of training observations in each terminal region (default, $kd = 10$).

call $setdtf(dtf)$: dtf = trimming factor used in splitting ($0 < dtf \leq 0.5$) (default, $dtf = 0.25$).

call $setlin(lin)$: lin = flag indicating splitting strategy and local approximator (default, $lin = 2$).

$lin = 0 \Rightarrow$ local constant model / axis oriented splits (only).
$lin = 1 \Rightarrow$ local linear model / axis oriented splits (only).
$lin = 2 \Rightarrow$ local constant model / linear combination splits.
$lin = 3 \Rightarrow$ local linear model / linear combination splits.

call $setrdg(rdg)$: rdg = ridge (weight decay) parameter used with local linear fitting and linear combination splits. A good value depends upon the degree of collinearity among the input variables over the training data (default, $rdg = 0.01$).

call $new(0)$: used only if DART is called more than once during execution of the same program. The argument has no meaning. Calling *new* signals that either the training data, or the parameters lin and/or rdg, have changed, and DART must reinitialize (sort data on all inputs and recompute global model). If these parameters and the data $x,\ y, w$ have not been changed then computation can be saved by not calling *new* before subsequent calls to DART.

2.2. Prediction

With eager training there are two strategies available for predicting future output values $\hat{y}$ given a prediction point $\mathbf{x}$. The first (described in Friedman (1996a)) is to use the local approximator corresponding to the region of the cover in which $\mathbf{x}$ is most centered. This is accomplished by the subroutine call

$$call\ ans(ni, xp, itr, rtr, trm, yh)$$

Input:

ni = number of inputs (same as in call to DART).
$xp(ni)$ = input values for prediction point $\mathbf{x}$.
itr, rtr, trm = output from DART.

Output:

yh = predicted output value $\hat{y}$.

This call can be repeated for a many predictions as desired.

A second prediction strategy (not described in Friedman (1996a)) is to use a weighted average of all the local approximators (evaluated at $\mathbf{x}$) corresponding to all the regions that contain the point $\mathbf{x}$. The weight given to each region is proportional to the (scaled) distance from its center to the prediction point. This is accomplished by the subroutine call

$$call\ ave(ni, xp, itr, rtr, trm, yh)$$

where all arguments are the same as in the call to ANS. This second strategy (AVE) requires much more computation for each prediction than the first (ANS) since all terminal nodes containing the prediction point $\mathbf{x}$ must be visited rather than only the one in which it is most centered. However, it (AVE) can sometimes be more accurate especially when local linear fitting ($lin = 1, 3$) is used since it allows curvature information to be taken into account.

3. Lazy/hybrid training/prediction

With a purely lazy learning strategy there is little or no training in preparation for prediction. When a prediction $\hat{y}$ an input point $\mathbf{x}$ is needed the region in which it is most centered is constructed, the local approximator is fit to the training data in that region, and a prediction is made. This approach eliminates the (potentially large) computation required for training at the expense of (considerably) increased computation for each prediction. The hybrid ("prudent") strategy is a compromise in which some investment in training is made in exchange for faster prediction. A cover tree is constructed with fewer (larger) terminal nodes than will be used for final prediction. When a prediction (at $\mathbf{x}$) is needed this tree is searched for the region of the corresponding cover in which $\mathbf{x}$ is most centered. A lazy strategy (HYESS - see below) is then applied to the training data in this region for further refinement, producing the final (smaller) prediction region. The local approximator is then fit to in this smaller region and a prediction is made. The eager part of the lazy/hybrid strategy is accomplished by the subroutine call

$$call\ start(no, ni, x, y, w, itr, rtr, itrm, rtrm, dtrm, mxm, mxi, mxr, mxd, ms).$$

The first eight arguments and the last (ms) are the same as in the call to *dart* (Section 2). The next three ($itrm, rtrm, dtrm$) store the terminal node information necessary to make subsequent (lazy) predictions (with HYESS):

$itrm(mxi)$ = integer storage.

$rtrm(mxr)$ = real storage.

$dtrm(mxd)$ = double precision storage.

These arrays must be appropriately declared and dimensioned in the calling program. The quantities mxm and ms are the same as in the call to DART (Section 2). The next three (mxi, mxr, mxd) are the respective dimensions of $itrm$, $rtrm$, and $dtrm$ as declared in the calling program. Appropriate values depend upon the problem at hand, the parameter values controlling tree construction, and (most importantly) the number of terminal nodes produced. These values should be set as large as possible consistent with available memory. If the storage provided for these arrays is found to be insufficient by the program (during tree construction) an appropriate diagnostic message is issued and execution terminates. If this happens, one can increase the storage allocated to the offending array (if possible). Alternatively, one can reduce the size of the tree by increasing the number of observations kd in each terminal node ($call\ setdtn(kd)$) or by increasing the trimming factor dtf ($call\ setdtf(dtf)$) used to construct the tree (see Section 2). The minimum size of all of these dimensions actually required by START for the problem can be obtained (after return from START) by the subroutine call

$$call\ stsize(mtr, mtim, mtrm, mtdm).$$

All quantities are output; mtr is the same as in $dtsize$ (Section 2), and the next three quantities are the actual minimum dimension size requirements (respectively) of the $itrm$, $rtrm$, and $dtrm$ arrays.

All of the parameters controlling DART tree construction described in Section 2 affect START in the exactly same way. They can be set (changed) by the appropriate subroutine calls (as described there) before the call to START.

3.1. Prediction

With the lazy/hybrid approach there is only one available strategy for prediction. That is, use the approximator corresponding to the region of the (final) cover in which the prediction point $\mathbf{x}$ is most centered [Friedman (1996a)]. This is accomplished by first calling START to construct an initial cover (tree). The size of this tree (2.1) is controlled by the values of the parameters kd and dtf (Section 2). If kd is set to the total training sample size ($kd = no$) then there is only one region covering the entire

input space and a purely lazy prediction strategy is invoked. Setting $kd < no$ implements a hybrid strategy. In either case individual predictions are subsequently made by the subroutine call

$$call\ hyess(xp, no, ni, x, y, w, itr, rtr, itrm, rtrm, dtrm, yh, mt)$$

Input:

$xp(ni)$ = input values of the prediction point $\mathbf{x}$.

no, ni, x, y, w = same as input to START.

$itr, rtr, itrm, rtrm, dtrm$ = output from START.

Output:

yh = predicted value $\hat{y}$.

Workspace:

$mt((2 \cdot ni + 3) \cdot no + 2 \cdot ni + 4)$ = integer scratch array.

3.2. Parameters

In addition to the parameters that control the tree building strategy for START (Section 2.1) there are others that apply uniquely to HYESS. These are set to default values in the program and can be changed by subroutine calls with the new values as arguments. There routines are:

call sethyn(kh): kh = number of training observations in each (final) terminal region ($0 < kh \leq kd$). Setting $kh = kd$ implements a purely eager strategy in a (much) less efficient way than using DART/ANS (Section 2) (default, $kh = 10$).

call sethtf(htf): htf = trimming factor used for lazy splitting ($0 < htf \leq dtf$). Note that setting $htf < dtf$ causes a smaller trimming factor to be used for the lazy part of the hybrid strategy. This is reasonable especially for large training samples (default, $htf = 0.1$).

call setknn(knn): knn = flag for including an Euclidean distance input variable [Friedman (1994)]:

$knn = 0 \Rightarrow$ no distance variable.

$knn = 1 \Rightarrow$ add distance variable.

If $knn = 1$ then the (weighted) Euclidean distance (squared) from the prediction point $\mathbf{x}$ to each training point $\mathbf{x}_i$

$$d_i^2 = \sum_{j=1}^{ni} [(x_j - x_{ij})/s_j]^2 \tag{3.1}$$

is computed and used as an additional input predictor to participate in the splitting. The scale factors $\{s_j\}_1^{ni}$ are the interquartile ranges of the respective inputs over the training data. Adding the distance (3.1) as a potential splitting variable can sometimes improve prediction accuracy at the expense of additional computing. For $knn = 1$ the dimension of xp in the call to *hyess* must be set to $xp(ni + 1)$ where $xp(1 \rightarrow ni)$ contains the input values of the prediction point $\mathbf{x}$ and $xp\,(ni + 1)$ is used as scratch workspace by the program. (Default: $knn = 0$.)

4. Model Selection

The principal problem dependent procedural (meta) parameter is the number of training observations K in each final terminal region. Others are lin which controls the splitting strategy / local model, and the ridge (weight decay) parameter rdg (Section 2.1). Good values for these parameters can be estimated through cross-validation. Given values for the others, XVALID can be called to provide an (automatic) estimate of the optimal value of K. It uses a lazy/hybrid strategy so a call to START (Section 3) must precede the call to XVALID. The calling sequence is

$$call\ xvalid(no, ni, x, y, w, itr, rtr, itrm, rtrm, dtrm, ks, erm, err, lm, mt).$$

Input:

$no, ni, x, y, w, itr, rtr, itrm, rtrm, dtrm$ = same as in call to $hyess$.

Output:

ks = estimated optimal number of observations in each terminal region.

erm = associated (minimized) average error.

$err(2, 40)$ = cross-validated error estimate as a function of number of training observations K in each terminal node.

$err(1, \cdot)$ = number of observations K.

$err(2, \cdot)$ = corresponding cross-validated error estimate.

lm = length of err array. (Note: $lm \leq 40$.)

Workspace:

$mt(2 \cdot no \cdot (ni + 1) + 2 \cdot (ni + 2) + no)$.

The value of K returned by XVALID (ks) is the one that minimizes the cross-validated estimate of future prediction error (erm). However, this value is seldom critical and there is usually a wide range of values for K that achieve nearly the same estimated error. From a computational perspective larger values of K are more desirable since they result in smaller trees. This in turn speeds up both training and prediction. By examining the cross-validated error estimate as a function of K (err) one can often choose a value for $K > ks$ that achieves close to the same estimated average error, thereby (often considerably) reducing computation without sacrificing much (if any) actual predictive accuracy.

4.1. Parameters

In addition to the parameters that control various aspects of the learning/prediction strategies (above) there are two more that are unique to cross-validation. They are set to default values in the program and can be changed by appropriate subroutine calls (before calling XVALID) with the new value as the argument.

call setcls(thr): thr = flag for prediction mode (default, $thr = 0$).

$thr \leq 0 \Rightarrow$ regression; cross-validate average absolute prediction error $ave\,|y - \hat{y}|$.

$thr > 0 \Rightarrow$ (two-class) classification. In this case the output value y is assumed to take on only two values $y \in \{0, 1\}$. The program (XVALID) uses the prediction rule $\tilde{y} = 1(\hat{y} > thr)$ as an output prediction, where $\hat{y}$ is the usual (regression) estimate, and cross-validates the average misclassification error rate $ave\,1(\tilde{y} \neq y)$.

call setkxv(kx): kx = computation reduction factor for cross-validation (default, $kx = 1$). The error is averaged over (removing) every kxth training observation. Setting $kx = 2$ averages over half of the data (every other observation) whereas $kx = 3$ averages over 1/3 of the data, etc. Setting $kx > 1$ can substantially reduce computation for cross-validation without (generally) increasing actual subsequent prediction error.

5. Internal storage

In addition to the storage arrays set up by the user in the main (calling) program, there are a variety of additional arrays internal to the program that are set up at compile time. The dimensions of these internal arrays are set in *parameter* statements in the subroutines where they are used. If during tree construction storage for one of these internal arrays is exceeded, a diagnostic message is issued and execution stops. The message indicates the name and internal value of the offending dimension parameter, as well as the name of the subroutine where its corresponding *parameter* statement is located. If this happens one can increase the value of the corresponding dimension parameter and recompile the program.

6. Examples

This section provides some simple illustrative example programs for invoking DART/HYESS in various modes. They can be used as starting points for analysis. However, it should be kept in mind that these illustrations are especially simple and far from exhaust the potential flexibility that can be achieved through more imaginative programming. In all cases the program calls a subroutine (DATA) that is assumed to provide the training data (x, y, w) in the respective arrays upon return.

6.1. Eager training (DART)

This program constructs a cover tree and writes it to disk for subsequent prediction. All parameters (Section 2.1) are left at their default values.

```
c set up size of the problem
      parameter (no=500, ni=10, mxm=100000, mxt=100000)
c dimension relevant arrays
      integer itr(2,mxm), ms((no+1)*(2*ni+1)+1)
      real x(no,ni+3), y(no), w(no), rtr(3,mxm), trm(mxt)
c read in data
      call data(x, y, w)
c build cover tree
      call dart(no, ni, x, y, w, mxm, mxt, itr, rtr, trm ,ms)
c obtain size of tree and terminal node storage
      call dtsize(mtr, mtm)
c open file for tree storage
      open(1, file='cover.tre', status='unknown')
c write size parameters
      write(1,*) mtr, mtm
c write cover tree
      write(1,*) (itr(1,j), itr(2,j), j=1,mtr)
      write(1,*) ((rtr(i,j), i=1,3), j=1,mtr)
```

```
c write terminal node information
      write(1,*) (trm(j), j=1,mtm)
c all done
      stop
      end
```

6.2. Eager prediction (ANS/AVE)

This program uses the cover tree produced by the preceding one (Section 6.1) to predict the output value $\hat{y}$ at the origin $\mathbf{x} = (0, \cdots, 0)$, using both the local predictor in which $\mathbf{x}$ is most centered (ANS), and the (weighted) average of the predictors associated with all regions containing $\mathbf{x}$ (AVE).

```
c set up storage for tree and local approximators
      parameter (ni=10, mxm=100000, mxt=100000)
      integer itr(2,mxm)
      real rtr(3,mxm), trm(mxt)
c set up and store prediction point
      real xp(ni) /ni*0.0/
c read in cover tree
      open(1, file='cover.tre', status='old')
      read(1,*) mtr, mtm
      read(1,*) (itr(1,j), itr(2,j), j=1,mtr)
      read(1,*) ((rtr(i,j), i=1,3), j=1,mtr)
      read(1,*) (trm(j), j=1,mtm)
c make predictions
      call ans(ni, xp, itr, rtr, trm, yhc)
      call ave(ni, xp, itr, rtr, trm, yha)
c write out predictions
      write (6,'("predictions at origin (ans, ave) =",2g12.4)') yhc, yha
c all done
      stop
      end
```

Note that multiple predictions (at other points) can be made by simply repeating the call to ANS and/or AVE with *xp* set to the subsequent input values for each such prediction point $\mathbf{x}$.

6.3. Hybrid training

This program constructs and writes to disk a partial cover tree with 4096 terminal regions to be used by HYESS for subsequent prediction (Section 6.4). It also changes the default settings of some of the parameters that control tree construction.

```
c set up size of problem and dimension arrays
      parameter (no=500, ni=10, mxm=5000, mxi=4000000, mxr=60000, mxd=600000)
      integer itr(2,mxm), itrm(mxi), mt((no+1)*(2*ni+1)+1)
      real x(no,ni+3), y(no), w(no), rtr(3,mxm), rtrm(mxr)
      double precision dtrm(mxd)
c read in training data
      call data(x, y, w)
c change (some) parameter settings
      call setlin(3)
      call setdtf(0.2)
c set K for 4096 terminal nodes
      call setdtn(int(no/4096.0**(alog(1.25)/alog(2.0))))
c build (partial) cover tree
      call start(no, ni, x, y, w, itr, rtr, itrm, rtrm, dtrm, mxm, mxi, mxr, mxd, mt)
c obtain tree size and terminal region storage
      call stsize(mtr, mtim, mtrm, mtdm)
c open file for tree storage
      open(1, file='cover.tre', status='unknown')
c write size parameters
      write(1,*) mtr, mtim, mtrm, mtdm
c write the cover tree
      write(1,*) (itr(1,j), itr(2,j), j=1,mtr)
      write(1,*) ((rtr(i,j), i=1,3), j=1,mtr)
c write terminal node information
      write(1,*) (itrm(j), j=1,mtim)
      write(1,*) (rtrm(j), j=1,mtrm)
      write(1,*) (dtrm(j), j=1,mtdm)
c all done
      stop
      end
```

6.4. Hybrid prediction

This program reads in the partial cover tree produced by the preceding one (Section 6.3) to predict the output value $\hat{y}$ at the origin $\mathbf{x} = (0, \cdots, 0)$ using HYESS. Note that in lazy/hybrid prediction the original training data is needed as well as the cover tree. Also, the parameter *lin* (Section 2.1) must be set to the same value as was used to construct the tree.

```
c set up size of problem and dimension arrays
      parameter (no=500, ni=10, mxm=5000, mxi=4000000, mxr=60000, mxd=600000)
      integer itr(2,mxm), itrm(mxi), mt(no*(2*ni+3)+2*ni+4)
      real x(no,ni+3), y(no), w(no), rtr(3,mxm), rtrm(mxr)
      double precision dtrm(mxd)
c set up and store prediction point
      real xp(ni) /ni*0.0/
c read in cover tree
      open(1, file='cover.tre', status='old')
      read(1,*) mtr, mtim, mtrm, mtdm
      read(1,*) (itr(1,j), itr(2,j), j=1,mtr)
      read(1,*) ((rtr(i,j), i=1,3), j=1,mtr)
      read(1,*) (itrm(j), j=1,mtim)
      read(1,*) (rtrm(j), j=1,mtrm)
      read(1,*) (dtrm(j), j=1,mtdm)
c read in training data
      call data(x, y, w)
c change parameter settings
      call setlin(3)
      call setdtf(0.1)
      call sethyn(2*ni)
c make prediction
      call hyess(xp, no, ni, x, y, w, itr, rtr, itrm, rtrm, dtrm, yh, mt)
c write out answer
write(6,'(" answer at origin =",g12.4)') yh
c all done
      stop
      end
```

Note that multiple predictions can be made by simply repeating the call to HYESS with *xp* set to the subsequent input values for each such prediction point $\mathbf{x}$.

6.5. Model selection

This program illustrates the use of XVALID for estimating an optimal value for K the number of observations in each terminal node. Here a purely lazy strategy is used so that storage for the tree and terminal node information need not be large.

```
c set up storage
      parameter (no=500, ni=10, mxm=10, mxi=20000, mxr=200, mxd=200)
      integer itr(2,mxm), itrm(mxi), mt(2*no*(ni+1)+2*(ni+2)+no)
      real x(no,ni+3), y(no), w(no), rtr(3,mxm), rtrm(mxr), err(2,40)
      double precision dtrm(mxd)
c read in data
      call data(x, y, w)
c change some parameter settings
      call setlin(3)
      call setdtf(0.1)
      call sethtf(0.1)
c set up start for purely lazy strategy (one treminal node)
      call setdtn(no)
c set xvalid for minimum K to be considered
      call sethyn(ni+1)
c initialize xvalid
      call start(no, ni, x, y, w, itr, rtr, itrm, rtrm, dtrm, mxm, mxi, mxr, mxd, mt)
c cross-validate with 1/2 of the data
      call setkxv(2)
      call xvalid(no, ni, x, y, w, itr, rtr, itrm, rtrm, dtrm, ks, erm, err, lm, mt)
c write out estimated optimal solution
      write(6,'(" optimal K =",i4, " with cross-validated error", g12.4)') ks, erm
c write out error as a function of K
      write(6,'("       K             error")')
      do 1 k=1,lm
      write(6,'(" ", f5.0,"       ", g12.4)') err(1,k), err(2,k)
    1 continue
c all done
      stop
```

end

If the value chosen for K (kc) is large enough so that purely eager training is possible one can *call* $setdtn(kc)$ and then call DART (Section 2) to obtain a complete cover tree. Predictions can then be made by calls to ANS or AVE. If not, then a hybrid strategy must be used by *call* $setdtn(kd)$, with $kd > kc$, and then calling START (Section 3) to produce a partial cover tree. Subsequent predictions are accomplished by *call* $sethyn(kc)$ before the calls to HYESS to make individual predictions.

References

[1] Friedman, J. H. (1994). Flexible metric nearest neighbor classification. Technical Report, Dept. of Statistics, Stanford University. (ftp://stat.stanford.edu/pub/friedman/flexmet.ps.Z)

[2] Friedman, J. H. (1996a). Local learning based on recursive covering. Technical Report, Dept. of Statistics, Stanford University. (ftp://stat.stanford.edu/pub/friedman/dart.ps.Z)

[3] Friedman, J. H. (1996b). DART/HYESS Fortran program.
(ftp://stat.stanford.edu/pub/friedman/progs/dart.for.Z)